Singular Value Decomposition: An Introduction with Applications to PCA and Machine Learning

March 2025

This is a link to the **Table of Contents**

Contents

I Preliminaries 4

1 Inner Products 4
- 1.1 Projection . . . 6

2 Quick Reference 7
- 2.1 Types of Matrices . . . 7
- 2.2 Definitions . . . 7

II Matrix Decomposition 8

3 Eigendecomposition 8
- 3.1 Special case: Spectral Decomposition . . . 9

4 Singular Value Decomposition 10
- 4.1 Existence of Singular Values . . . 12
- 4.2 Implementation and Sensitivity . . . 13
  - 4.2.1 Computational Pitfalls . . . 14
  - 4.2.2 Recovering Eigenvalues . . . 15
- 4.3 Low Rank Approximation . . . 15

III Compression 17

5 Image Compression 17

6 Video Compression 18

7 Sound Compression 19

8 Code 20
- 8.1 Image . . . 20
- 8.2 Video . . . 21
- 8.3 Sound . . . 22

IV Principal Component Analysis 24

9 Statistics 24
- 9.1 Signals, Noise, and Redundancy . . . 25
- 9.2 Covariance . . . 25

10 Principle Components 27
- 10.1 Assumptions . . . 27
- 10.2 Overview . . . 27
  - 10.2.1 Simplified Calculation . . . 28
  - 10.2.2 Feature Selection . . . 28
  - 10.2.3 Implementation . . . 29
- 10.3 Examples . . . 29

10.3.1 PCA of a Small Dataset 29
10.3.2 PCA of a Large Dataset 31

11 Exercises **33**

V Code **34**

12 Gram-Schmidt **34**

13 PCA of a Small Dataset **35**
13.1 First Iteration 37

14 PCA of a Large Dataset **39**

VI Bibliography **42**

This is a link to the **Table of Contents**

Part I
Preliminaries

Introduction

This part serves to introduce concepts that motivate what follows as well as provide a reference for material that may not show up in an introductory linear algebra course. Because there is no better place to state it, we shall state it here: in a finite dimensional vector space, every linear transformation has a matrix representation! We will always assume that we are in a finite dimensional vector space and, in our examples, that $\mathbb{R}$ is the underlying field[1].

Note 1. The notation for vectors will differ between authors; some texts present them in boldface as $\mathbf{x}$, others use the arrow $\vec{x}$, and others do nothing at all. We elect to use boldface when there may be ambiguity.

1 Inner Products

A vector space contains objects that behave linearly, where the typical operations of addition and (scalar) multiplication of vectors behave the same regardless of the order in which they are done - this is a more intuitive way of saying that vector addition and scalar multiplication are both linear maps, which can be generalized to linear transformations; here, a linear map goes between vector spaces. It seems natural that there should be a similar type of map from a vector space to its underlying field, and in fact there is: the inner product.

Definition I.1 (Inner Product)**.** *Let V be a vector space over a field F. The function defined by*

$$\langle \cdot, \cdot \rangle : V \times V \longrightarrow F$$

*is called an **inner product** of V if and only if, for all $x, y, z \in V$ and $a, b \in F$,*

1. $\langle ax + by, z \rangle = a\langle x, z \rangle + b\langle y, z \rangle$
2. $\overline{\langle x, y \rangle} = \langle y, x \rangle$
3. $\langle x, x \rangle \geq 0$, *with equality if and only if* $x = 0$

Example. [The Dot Product] Let $x, y \in \mathbb{R}^n$ and denote their components x_i and y_i for $1 \leq i \leq n$. Then $\langle x \cdot y \rangle = \sum_{i=1}^{n} x_i y_i$; the sum of the products of corresponding components. ☼

Example. [Integral Product] Let $C(I)$ denote the set of real-valued continuous functions with domain $I = [a, b]$; the proof that this is a vector space is omitted. Let $f, g \in C(I)$, and define the inner product as

$$\langle f, g \rangle = \int_a^b f(x)g(x)\,dx$$

☼

[1] To pique the reader's interest, it is also worth mentioning that there do exist many important infinite-dimensional spaces, particularly function spaces, and that vector spaces in general can be endowed other structures as well. Such examples are polynomial rings, fields, Lie Algebras, and topological vector spaces (which include function spaces). Additionally, there are vector spaces over finite fields. These are most commonly studied in the context of field extensions rather than in a linear algebra sense, and perhaps with applications to algebraic geometry.

It is important to know that inner products are not unique, as any given vector space can have many different inner products. Informally, one can think of the inner product between two vectors as a measure of how "alike" they are relative to their basis. Any two vectors whose inner product is zero are said to be **orthogonal**; for example, in $\mathbb{R}^2$ with the standard basis, any vectors of the form $(x, 0)$ and $(0, y)$ are at a $90°$ angle and their dot product is zero. We say that a vector space equipped with an inner product is called an **inner product space**. As the existence of multiple inner products implies, one vector space can give rise to multiple inner product spaces, each with different structures imposed by the particular inner product used to define it. The following properties follow immediately from the definition given above.

Theorem P.1 (Inner Product Space)**.** Suppose V is a vector space with an inner product $\langle \cdot, \cdot \rangle$. Then, for all $x, y, z \in V$ and $a \in F$,

1. $\langle x, ay \rangle = \bar{a}\langle x, y \rangle$
2. $\langle x, 0 \rangle = \langle 0, x \rangle = 0$
3. If $\langle x, y \rangle = \langle x, z \rangle$ for all $x \in V$, then $y = z$.

In thinking of inner products as quantifying the comparison of vectors, we are also able to quantify just how "big" a vector is relative to the inner product: the norm, which is a natural extension of the concept[2]. It follows that every inner product space is a normed vector space.

Definition I.2 (Norm)**.** *Let V be an inner product space and let $x \in V$. Then $\|x\| = \sqrt{\langle x, x \rangle}$ is called the **norm** of x. If $\|x\| = 1$, then we say x is a **normal** or **unit** vector.*

Example. [Euclidean Space] Let $x \in \mathbb{R}^n$, with components x_i. The Euclidean (also called the quadratic, ℓ^2, or square) norm is

$$\|x\|_2 = \left(\sum_{i=1}^{n} (x_i)^2 \right)^{1/2}$$

☼

Example. [Function Spaces] Let $f \in C(I)$. Respectively, the square and supremum norms are given by

$$\|f\|_2 = \left(\int_a^b |f(x)|^2 \, dx \right)^{1/2} \quad \text{and} \quad \|f\|_\infty = \sup \left\{ \, |f(x)| : x \in [a, b] \, \right\}$$

☼

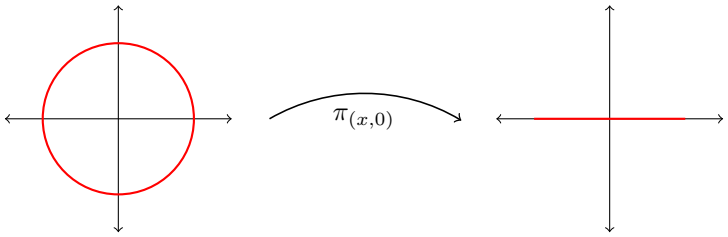
Let's end this section on an interesting note. If we think of $C(I)$ as a metric space, it is not complete in the square norm. Defined below, the sequence f_n converges to a function that is not continuous; however, in the supremum norm, this is not the case.

$$f_n(x) = \begin{cases} 0, & x \in I_1 = \left[0, \frac{1}{2} - \frac{1}{n}\right) \\ n\left(x - \frac{1}{2} + \frac{1}{n}\right), & x \in I_2 = \left[\frac{1}{2} - \frac{1}{n}, \frac{1}{2}\right) \\ 1, & x \in I_3 = \left[\frac{1}{2}, 1\right] \end{cases}$$

[2]To be precise, F must be a *normed field* for this to occur. There is a theorem that states every normed field is either a subfield of $\mathbb{C}$ or that it has the non-Archimedean Property $|x + y| \leq \max\{|x|, |y|\}$ for all $x, y \in F$; while a normed space *can* be defined over a non-Archimedean field, the differences between such a thing and what we're familiar with are probably worth giving it a new name entirely and treating them as being different.

1.1 Projection

A projection is a linear transformation that does exactly what it sounds like and is perhaps very intuitive - it is similar to the idea of shining a light behind an object and observing the shadow it leaves on a surface; for example, the projection of the unit circle in $\mathbb{R}^2$ onto the x-axis is the closed interval $[-1, 1]$. Note that this is not unique: shifting the circle along the y-axis yields the same image under the transformation! Below is a picture representing this concept. Without further ado, let's provide a formal definition.



Definition I.3. *Let V be an inner product space and let $\{v_1, ..., v_k\}$ be a basis for a subspace $W \subset V$. Define $\pi : V \to W$ as follows:*

$$\pi(x) = \sum_{i=1}^{k} \frac{\langle x, v_i \rangle}{\langle v_i, v_i \rangle} v_i$$

*Then π is said to be the **projection map** from V onto W. Furthermore,*

1. $\pi \circ \pi = \pi$
2. $\pi(x) = x$ *if and only if* $x \in W$, *and*
3. $\langle x - \pi(x), w \rangle = 0$ *for all* $w \in W$

*If $\langle \pi(x), y \rangle = \langle x, \pi(y) \rangle$ for all $x, y \in V$, then π is an **orthogonal projection**.*

Projection maps have many uses, the most notable of which is the algorithm defined below. This takes a set of vectors and uses projection to transform them into an orthonormal set; that is, each of the vectors are normal and they are all orthogonal to each other. Computational efficiency is greatly improved when data has been orthonormalized!

Definition I.4 (The Gram-Schmidt Process)**.** *Let V be a vector space with $\{x_i\}_{i=1}^n$ a set of linearly independent vectors. Set $x_0 = v_0$. The **Gram-Schmidt Process** is the following algorithm*

$$v_k = x_k - \sum_{i=1}^{k-1} \frac{\langle x_k, v_i \rangle}{||v_i||^2} v_i$$

and generates a set of vectors $\{v_i\}_{k=1}^n$, all of whom are orthogonal.

The essence of the process is that we subtract from each vector x_k the "relatedness" of everything that we've already done; from here, to achieve normality is a matter of scaling each vector appropriately.

Exercise 1. Let A be a basis for $\mathbb{R}^3$ containing vectors x, y, and z whose components are given below. Using your preferred language, construct an orthogonal basis.

$$\begin{aligned} x_1 &= 5, \quad x_2 = 2, \quad x_3 = 1.5 \\ y_1 &= -1, \quad y_2 = -4, \quad y_3 = 1 \\ z_1 &= 3, \quad z_2 = 0, \quad z_3 = 2 \end{aligned}$$

This is a link to the **Table of Contents**

2 Quick Reference

2.1 Types of Matrices

Definition M.1 (Symmetric)**.** A matrix M is said to be **symmetric** if and only if $M^T = M$.

Definition M.2 (Normal)**.** A matrix M is said to be **normal** if and only if it satisfies $MM^* = M^*M$. Over $\mathbb{R}$, a symmetric matrix is normal by definition.

Definition M.3 (Definite)**.** Let M be a square matrix. M is said to be **positive definite** if and only if for each eigenvalue λ, we have $\lambda > 0$. If $\lambda \geq 0$, we say that M is **positive semi-definite**; in particular, if and only if the real number $x^T Mx$ is not negative for all vectors $x \in \mathbb{R}^n$.

Definition M.4 (Orthogonal/Unitary)**.** A matrix M is said to be **orthogonal** if and only if its column vectors are all orthonormal; that is, $M^{-1} = M^T$. For $U \in \mathbb{C}^{n\times n}$, it is **unitary** if and only if $U^{-1} = U^*$. In either case, $MM^T = UU^* = I$.

2.2 Definitions

Definition P.1 (Trace of a Matrix)**.** Let $M \in \mathbb{C}^{m\times n}$. The **trace** of M, denoted trace(M) or tr(M), is the sum of the entries along the main diagonal. In symbols,

$$\text{tr}(M) = \sum_{i=1}^{\max\{m,n\}} m_{ii}$$

Definition P.2 (Rank of a Matrix)**.** The **rank** of a matrix M, denoted Rank(M), is the dimension of the vector space spanned by its columns; that is, the number of linearly independent column vectors of M.

Definition P.3 (Adjoint of a Matrix)**.** Let M be a complex-valued matrix. The **adjoint** of M is denoted M^* and obtained by the following (commutative) map: for $a, b \in \mathbb{R}$ and $m_{i,j} \in M$ where $m_{i,j} = a + bi$,

$$m_{i,j} = a + bi \quad \mapsto \quad a - bi = m^*_{j,i}$$

That is, M^* is the transpose of M whose imaginary parts have changed sign. Over $\mathbb{R}$, this is simply the transpose.

Definition P.4 (Unitarily Invariant Norm)**.** Let $\|\cdot\|$ be a norm on $\mathbb{C}^{m\times n}$. We say that $\|\cdot\|$ is **unitarily invariant** if and only if, for all $M \in \mathbb{C}^{m\times n}$, we have

$$\|UMV\| = \|M\|$$

for all unitary $U \in \mathbb{C}^{m\times m}$ and $V \in \mathbb{C}^{n\times n}$.

Part II
Matrix Decomposition

Introduction

A technique used throughout mathematics is to rethink a problem in a different perspective or field, solve it or reduce it there, then convert it back into the original framework - for example, problems involving irreducible polynomials in field theory are difficult to solve but the group representations of fields allows these problems to be solved in terms of permutation groups. In linear algebra, we have the ability to change the basis of a vector space into a more elegant form in order to study a particular transformation. Matrix decomposition is of surpassing importance in numerical linear algebra; computations are quick to become resource intensive as the dimensions of matrices increase, and it is not always possible to reduce them. By decomposing them into their parts, we are able to quickly identify key properties of the transformations they represent and determine if computation is feasible[3]. The importance of computational efficiency cannot be emphasized enough and will be frequently remarked on.

3 Eigendecomposition

Eigendecomposition is a fundamental concept in linear algebra that refers to the factorization of a square matrix into a canonical form of eigenvalues and eigenvectors. Students studying mathematics should be aware that certain matrices, whether over $\mathbb{R}$ or $\mathbb{C}$, can be diagonalized. Let A be an $n \times n$ matrix. Recall the formula for a change of basis, where P and P^{-1} are the change of basis matrices:

$$A = PDP^{-1}$$

Supposing that D is a diagonal matrix of the form

$$D = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n).$$

we say that A is **diagonalizable**. If A has n eigenvectors, and if the dimension of the vector space is n, then the eigenvectors of A form an ordered basis - that is, for $i \in \{1, \ldots, n\}$, the eigenvectors v_i of A form the columns of P while the corresponding eigenvalues λ_i appear as the diagonal entries of D. This process of factorization is known as **eigendecomposition**.

Example. Let's find the eigendecomposition of a 2×2 matrix. Consider the real 2×2 matrix A, defined below. We want to diagonalize A by finding a non-singular matrix Q such that $Q^{-1}AQ = D$, where $Q, D \in \mathbb{R}^{2\times 2}$. Let a, b, c, d be the entries of Q and λ_i the eigenvalues on the diagonal of D; we have

$$A = \begin{pmatrix} 1 & 0 \\ 1 & 3 \end{pmatrix}, \qquad D = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

Multiplying both sides of the equation on the left by Q, we obtain $AQ = QD$; expanding this yields

$$\begin{pmatrix} 1 & 0 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

Let $\mathbf{q_1}$ and $\mathbf{q_2}$ be the column vectors of Q. The above results in two simultaneous equations $A\mathbf{q_1} = \lambda_1\mathbf{q_1}$ and $A\mathbf{q_2} = \lambda_2\mathbf{q_2}$, which we can rewrite as $(A - \lambda_i I)\mathbf{q_i} = \mathbf{0}$. Now we want solve the

[3] For example, the Higgs Boson was announced "discovered" on July 4, 2012, but in reality the announcement was about a particle in the expected mass range. It took until March of the following year for researches to be sure that a Higgs Boson had actually been found, processing over 1 petabyte of data.

characteristic polynomial, found by $\det(A - \lambda I) = 0$. Computing the determinant,

$$\begin{vmatrix} 1-\lambda_1 & 0 \\ 1 & 3-\lambda_2 \end{vmatrix} = (1-\lambda_1)(3-\lambda_2) = 0 \quad \longrightarrow \quad \lambda_1 = 1, \quad \lambda_2 = 3$$

Thus, the diagonal matrix from the eigendecomposition of A is

$$D = \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix}$$

We find the eigenvectors by substituting $\lambda_1 = 1$ into $(A - \lambda_1 I)\mathbf{q_1} = 0$:

$$\begin{pmatrix} 1-1 & 0 \\ 1 & 3-1 \end{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} a \\ c \end{pmatrix} = 0$$

which results in $a = -2c$, meaning c is a free variable; moreover, we apply (but omit) the same process with $\lambda_2 = 3$. Thus the eigenvectors are nonzero scalar multiples of

$$\mathbf{q_1} = \begin{pmatrix} -2 \\ 1 \end{pmatrix} \quad \text{and} \quad \mathbf{q_2} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

which are the columns of Q. It is trivial to calculate the inverse, so we find that the eigendecomposition of A is

$$\begin{pmatrix} 1 & 0 \\ 1 & 3 \end{pmatrix} = \begin{pmatrix} -2 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 3 \end{pmatrix} \begin{pmatrix} -1/2 & 0 \\ 1/2 & 1 \end{pmatrix}$$

☼

Numerical algorithms have a property called **numerical stability**, where we want algorithms to be as stable as possible; however, numerical methods have limitations. For example, **catastrophic cancellation** occurs when values are sufficiently close together because errors made when rounding compound as successive operations are performed. For example, suppose we have 13.49L and 13.51L of some liquid: while their volumes differ by 0.02L, rounding to 13L and 14L appears to make them differ by 1L. These issues are exacerbated with multiplication and exponentiation, especially when it comes to matrix operations.

Eigenvalues are called ill-conditioned when they are sensitive to rounding or perturbation, which makes them susceptible to the limitations of numerical methods used in their calculation[4]. To give a general rule, matrices further from being normal will have more sensitive eigenvalues [1]. In particular, the matrix AA^T is very prone to these issues. The classic example is the Läuchli Matrix, where $0 < \epsilon \ll 1$, which we will explore later.

3.1 Special case: Spectral Decomposition

The Spectral Theorem is a special case of eigendecomposition and is arguably one of the most important results in linear algebra. As it provides a condition for when a matrix may be diagonalized, computations involving these matrices are greatly simplified as the products of diagonal matrices amount to simply multiplying corresponding terms on the diagonal. It also has utility in other fields of mathematics that make use of matrix algebra, which in turn lead to their own problems being greatly simplified.

First proven by Cauchy, the Spectral Theorem was generalized by von Neumann where it now ranks among the most important results of operator theory (the study of linear operators over function spaces)[5]. For our purposes, it is sufficient that we stick with matrices over $\mathbb{R}$ instead.

[4] See here for an example using MATLAB. The initial eigenvalues are represented by red stars.
[5] For those with unbounded curiosity, the 26 page paper found here is a full account of it and its proof.

Theorem 3.1 (Spectral Theorem)**.** *Let $A \in \mathbb{R}^{n\times n}$. The following are equivalent:*

1. *A is normal.*
2. *$\mathbb{R}^n$ has an orthonormal basis consisting of eigenvectors of A.*
3. *A can be diagonalized.*

Because every real valued symmetric matrix is normal, it follows that every symmetric matrix is diagonalizable. In practice, this means that a normal matrix A has the following eigendecomposition

$$A = Q\Lambda Q^T$$

where Q is an orthonormal matrix whose columns are the eigenvectors of A and Λ is a diagonal matrix of corresponding eigenvalues, and we say that A has a **spectral decomposition** when we may apply the Spectral Theorem.

4 Singular Value Decomposition

Singular Value Decomposition (SVD) is a more generalized matrix factorization technique and provides a geometric description of its properties. In particular, with SVD we can analyze and describe matrices that are pathological in nature - even non-square matrices can be decomposed; from a geometric standpoint, we have a rotation, scale, and another rotation.

Definition II.1. *Without loss of generality, suppose that $m \geq n$. Let $M \in \mathbb{C}^{m\times n}$ and* $\operatorname{Rank} M = r$*. The **Singular Value Decomposition of** M is a factorization of M as a product of three matrices:*

$$M = U\Sigma V^*$$

where $U \in \mathbb{C}^{m\times m}$ and $V \in \mathbb{C}^{n\times n}$ are unitary, $\Sigma \in \mathbb{C}^{m\times n}$ is rectangular diagonal, and V^* is the adjoint of V. When restricted to $\mathbb{R}^{m\times n}$, U and V are orthogonal and we write $V^* = V^T$. In addition,

- *The **singular values of M** are the entries of Σ on the diagonal, real valued, and are denoted $\sigma_{\mathbf{i}}$ for $i \leq r$; for all $r < j \leq n$, we set $\sigma_j = 0$.*
- *The **left singular vectors of M** are the column vectors of U, denoted $\mathbf{u_i}$, and form an orthonormal basis for $\mathbb{C}^n$.*
- *The **right singular vectors of M** are the column vectors of V, denoted $\mathbf{v_i}$, and form an orthonormal basis for $\mathbb{C}^m$.*

Note 2. Supposing that $n > m$, we have $M^T = U\Sigma V^*$. But then $M = V\Sigma^T U^*$, so we really may assume $m \geq n$ (i.e., at least as many rows as columns) without any loss of generality.

SVD is unique up to permutations of the columns of U and V, which is why it is standard practice to arrange the σ_i in decreasing order. From here on, when we mention SVD, we are referring to this arrangement; that way, we do not have to specify which SVD we refer to. It may be the case that $\operatorname{Rank}(M) < m$; that is, a column of M is not linearly independent and so at least one singular value with be zero. Letting $r = \operatorname{Rank}(M)$, the SVD can also be written as a linear combination:

$$M = \sum_{i=1}^{r} \sigma_i \mathbf{u_i} \mathbf{v_i}^*$$

Definition II.2 (Frobenius Norm)**.** *Denoted $\|\cdot\|_F$, the **Frobenius norm** of a matrix M the square root of the sum of the squared absolute values of its elements; in symbols,*

$$\|M\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|m_{ij}|^2}$$

Or, more simply,

$$\|M\|_F = \sqrt{\mathrm{tr}(MM^*)}$$

Definition II.3 (Spectral Norm)**.** *Denoted $\|\cdot\|_2$, the **Spectral norm** of a matrix M is*

$$\|M\|_2 = \sigma_1 = \sigma_{max}$$

Below is a representation of what the SVD of a 3×2 matrix M would look like in general. The vectors $\mathbf{u_i}$ are column vectors of length 3 and $\mathbf{v_j}$ are column vectors of length 2:

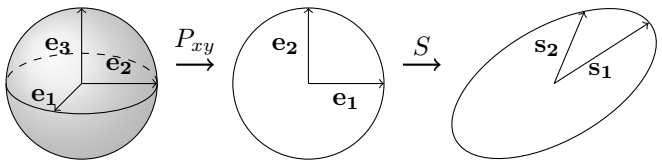
$$\begin{array}{ccccc} M & = & (\mathbf{u_1}|\mathbf{u_2}|\mathbf{u_3}) & \mathrm{Diag}(\sigma_1,\sigma_2) & (\mathbf{v_1^*}|\mathbf{v_2^*}) \end{array}$$

$$\begin{pmatrix} m_{1,1} & m_{2,1} \\ m_{1,2} & m_{2,1} \\ m_{1,3} & m_{2,3} \end{pmatrix} = \begin{pmatrix} u_{1,1} & u_{2,1} & u_{3,1} \\ u_{1,2} & u_{2,2} & u_{3,2} \\ u_{1,3} & u_{2,3} & u_{3,3} \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \overline{v_{1,1}} & \overline{v_{1,2}} \\ \overline{v_{2,1}} & \overline{v_{2,2}} \end{pmatrix}$$

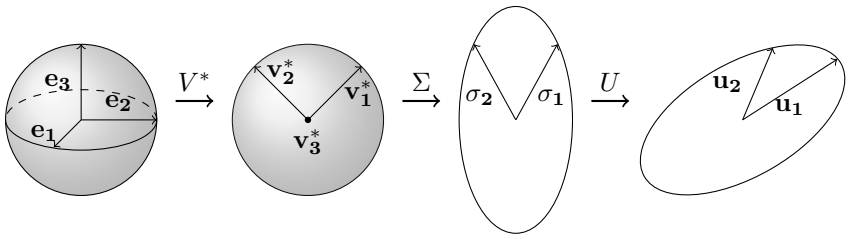
Example. Consider the following matrix:

$$M = \begin{pmatrix} \frac{21}{16} & \frac{15}{16} & 0 \\ \frac{3\sqrt{3}}{16} & \frac{9\sqrt{3}}{16} & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

We know that matrices represent linear transformations and we often describe their actions by using some reference shape. We also know that matrices do not factor uniquely, so let's interpret the transformation as a projection onto the xy-plane (or e_1 and e_2, in terms of standard basis) and a shear; that is, if $M = P_{xy}S$



On the other hand, with SVD, we can avoid shearing entirely by factoring M into three transformations - rotating, stretching and compressing, then rotating again - and glean much more information. The singular values determine by how much a shape is stretched or compressed and in what directions, as we see below.



Now let's examine how this actually factorizes. Let $\theta = -\pi/3$; then,

$$V^* = \begin{pmatrix} \cos(\theta) & -\cos(\theta) & 0 \\ -\sin(\theta) & -\sin(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} 0.75 & 0 & 0 \\ 0 & 1.5 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad U = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Here, we have $\sigma_1 = 0.75$, $\sigma_2 = 1.5$, and $\sigma_3 = 0$. Already, we can see that SVD encodes how the axes of the sphere are scaled. ☼

4.1 Existence of Singular Values

We will prove that singular values exist, but first it is important to remark this proof is not comprehensive; a complete proof requires other theorems, facts, and definitions that are definitely a departure from the point. Understanding the proof as presented is sufficient to motivate the intuition of how SVD works as well as serve as a loose guide to its implementation. In addition, as eigendecomposition is an integral part of the proof (which is in what we present), it is worth stressing the importance of eigenvalues is applied mathematics.

Our presentation of this proof is based off of Strang's work[6], and a reader may think that this proof is akin to someone climbing onto a roof using a ladder, kicking away said ladder, then shouting about magic or something - we feel that it is worth explaining the motivation. First, imagine that the SVD actually exists for some matrix. What must the decomposition look like? Since we have square matrices, what are the eigenvalues? In other words, his proof can be thought of as assuming that it exists, determining what it **would** look like if it did, then showing that it *actually* does. This is quite a useful technique in proving things, especially for things like proving (or disproving) the continuity of some function using the ϵ-δ definition of continuity, and especially when one develops intuition in a particular field. In a sense, this is like "working backwards".

Theorem 4.1 (Singular Value Decomposition)**.** *Let $M \in \mathbb{C}^{n\times m}$ with $n \geq m$. Then the SVD of M as described in II.1 actually exists.*

Proof. Observe that M^*M is symmetric, meaning that it is diagonalizable by an $m \times m$ unitary matrix V; moreover, it positive semi-definite, so we are justified in letting $\lambda_i = \sigma_i^2$ be its eigenvalues. Decomposing this matrix, we have

$$M^*M = VDV^* = \sum_{i=1}^{n} \lambda_i v_i v_i^* = \sum_{i=1}^{n} \sigma_i^2 v_i v_i^*$$

Let $r = \text{Rank}(M) = \text{Rank}(M^*M)$ and enumerate the σ_i such that $\sigma_1^2 \geq \sigma_2^2 \geq ... \geq \sigma_r^2$. Since V is unitary, its columns are orthonormal eigenvectors:

$$V = \left(\mathbf{v_1} \,\middle|\, \mathbf{v_2} \,\middle|\, ... \,\middle|\, \mathbf{v_r} \,\middle|\, \mathbf{v_{r+1}} \,\middle|\, ... \,\middle|\, \mathbf{v_m}\right)$$

Observe that the first r eigenvectors correspond to nonzero eigenvalues, meaning that the rest are an orthonormal basis for the zero-eigenspace (the nullspace of M^*M). Now for each σ_i we construct unit eigenvectors for MM^*, which is the key relation in SVD:

$$\mathbf{u_i} = \frac{1}{\sigma_i} M\mathbf{v_i}$$

In other words, if $\mathbf{v}$ is a unit eigenvector of M^*M with eigenvalue σ^2, then $\mathbf{u} = \frac{1}{\sigma} M\mathbf{v}$ is a unit eigenvector of MM^*. Let U_r be the $n \times r$ matrix with columns $\mathbf{u_i} = \frac{1}{\sigma} MV\mathbf{v_i}$. Observe that

[6] This is a link to his 1999 lecture at MIT.

1. V is the $m \times m$ matrix with columns $\mathbf{v_i}$
2. V_r are the first r columns of V
3. Σ_r is the $r \times r$ diagonal matrix with entries σ_i
4. Σ_r^{-1} has entries $\frac{1}{\sigma_i}$

which means $U_r = MV_r\Sigma_r^{-1}$, and can be rearranged to $U_r\Sigma_r = MV_r$. Now we multiply the adjoint of V_r on the right:

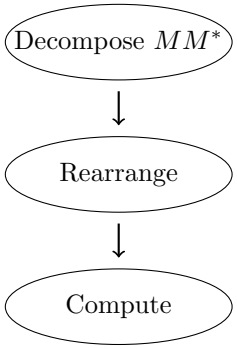
$$\begin{aligned} U_r\Sigma_r V_r^* &= MV_rV_r^* \\ &= MI_r = M \end{aligned}$$

Finally, we extend U_r to an $n \times n$ matrix U by adding $n - r$ orthonormal vectors as columns, extending Σ_r to Σ by adding $m - r$ columns of zeroes and $n - r$ rows of zeroes, and reconsider V in full. Then V and U are orthonormal bases for $\mathbb{C}^n$ and $\mathbb{C}^m$, respectively, and the relation $M = U\Sigma V^*$ is preserved. □

4.2 Implementation and Sensitivity

The process of decomposing a matrix by its singular values is demonstrated through the proof, but as careful understanding of it is not the point, it is the best practice to provide an explicit guide on how to actually do these calculations:

1. Compute MM^* and perform eigendecomposition; that is, $MM^* = VDV^*$. Then $\sigma_i = \sqrt{\lambda_i}$ are the singular values.
2. Rearrange D so that the eigenvalues are in descending order, and do the same to maintain correspondence with V and V^*.
3. Determine the rank of M, call it r. The first r columns of V are the right singular values $\mathbf{v_i}$, and the left singular values are $\mathbf{u_i} = \frac{1}{\sigma_i}M\mathbf{v_i}$.



It is often not necessary to use the full SVD of a matrix, especially when working with real-world data; there may be hundreds of rows and columns that make computation unfeasible or the results unreadable. **Compact SVD** eliminates the columns of U and rows of V^* that correspond to singular values equal to zero; that is, we have $M = U_r\Sigma_r V_r^*$ as in the proof.

We presented eigendecomposition by defining it, showing its utility, demonstrating an example, and finally explaining its limitations due to the sensitivity of eigenvalues. For singular values, so far we provided relevant definitions and an example with geometric motivation. There is one question remains: if eigenvalues are sensitive to perturbation, what about singular values? We do not prove the result, but it turns out that singular values are not sensitive to perturbations and that "close enough" is sufficient when we wish to approximate a particular matrix.

Theorem 4.2 (Perturbation Theorem for Singular Values)**.** *Let M and B be two $m \times n$ matrices with $B = M + E$. Let σ_i be the singular values of M and $\tilde{\sigma}_i$ the singular values of B, indexed in decreasing order. Then $|\tilde{\sigma}_i - \sigma_i| \leq \|E\|_2$ for each i.*

4.2.1 Computational Pitfalls

There is an apparent incongruity lurking in the background: the method that we have presented to find the singular values of a matrix uses eigenvalues, yet we have also established that eigendecomposition is susceptible to the eigenvalues being ill conditioned; furthermore, our implementation even commits the cardinal sin of *using* the eigenvalues of MM^* to find the singular values! If we use this method, then the example below demonstrates that our calculation of the singular values may be suspect.

Example. Let $0 < \epsilon \ll 1$. The Läuchli Matrix L is defined as

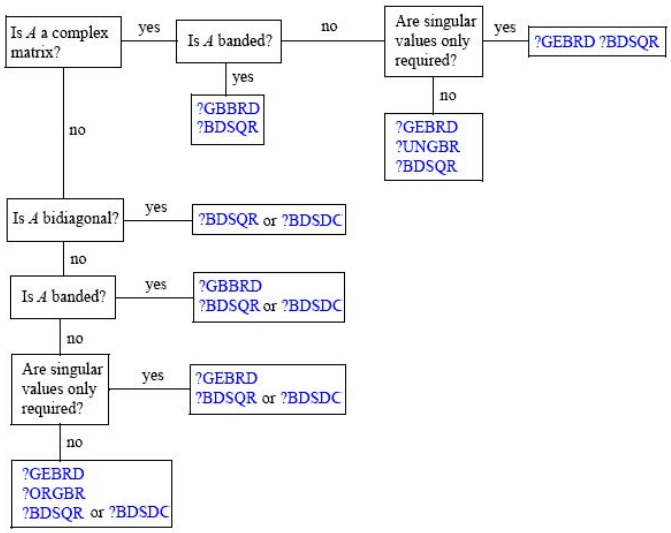
$$L = \begin{pmatrix} 1 & 1 & 1 \\ \epsilon & 0 & 0 \\ 0 & \epsilon & 0 \\ 0 & 0 & \epsilon \end{pmatrix}$$

Multiplying by L^T, we have

$$LL^T = \begin{pmatrix} 3 & \epsilon & \epsilon & \epsilon \\ \epsilon & \epsilon^2 & 0 & 0 \\ \epsilon & 0 & \epsilon^2 & 0 \\ \epsilon & 0 & 0 & \epsilon^2 \end{pmatrix}$$

Using WolframAlpha, the SVD of L yields terms of the form $3/\sqrt{3\epsilon^2+9}$ in V^T, while computing the eigendecomposition of LL^T will yield $1/3$ instead - a discrepancy caused by improper rounding. ☼

There are methods to calculate the singular values avoiding the usage of eigenvalues. For instance, below is the LAPACK schematic used by oneMKL [2].



It is of no use to provide this figure without context, so here is the general overview of what the particular classes of routines are actually doing:

1. The ?G routines reduce A into a bidiagonal matrix B if necessary.
2. The ?O and ?U routines generate and apply the orthogonal and unitary matrices if required.
3. The ?B routines form the SVD of B.

Embedded within these routines are QR methods. These are not without flaws as particular QR decompositions have their own limitations:

- The Gram-Schmidt process is incredibly easy to implement, but it may yield vectors that are not orthogonal due to rounding errors which defeats the purpose of using it.
- Householder transformations are the most simple of the stable methods, but as the Householder reflection algorithms require care to avoid catastrophic cancellation, they require greater computational bandwidth and are difficult to conduct in parallel.
- Givens rotations turn an element in the subdiagonal to zero, meaning that they will form the R matrix, and their concatenation forms the Q matrix. However, as the way to properly order the rows is challenging to determine, its implementation as an algorithm can become exceedingly difficult and is more useful when only a few off-diagonal elements are nonzero.

Because of these problems, the general advice is against implementing one's own SVD algorithm without having a stronger background in numerical linear algebra[7]. It is not difficult to create a suboptimal implementation that either takes too long to to run, is inaccurate, or whose results are simply wrong.

4.2.2 Recovering Eigenvalues

We may not be able to avoid working with the matrix MM^*; as it happens, this matrix is instrumental in the following sections, so we truly cannot afford to write it off and a stepping stone and put it out of mind. In fact, the eigenvalues of this matrix are also incredibly important! While we cannot reliably perform calculations with MM^*, it is a simple matter of using SVD as a tool to recover its eigenvalues.

Proposition 4.1. *Let $M \in \mathbb{C}^{m\times n}$ with singular values σ_i. Then the product of the SVD of M and M^* has factors containing the eigenvalues and eigenvectors of MM^*.*

Proof. Let $U\Sigma V^*$ be the SVD of M with singular values σ_i. It follows that the SVD of M^* is the adjoint of $U\Sigma V^*$, so $M^* = V\Sigma^* U^*$. Then, since V and V^* are orthonormal,

$$\begin{aligned} MM^* &= (U\Sigma V^*)\,(V\Sigma^* U^*) \\ &= U\Sigma\Sigma^* U^* \end{aligned}$$

Recall that the columns of U are the eigenvectors, which correspond to the values of the matrix Σ whose entries are the singular values σ_i. But since Σ is $m \times n$ and Σ^* is $n \times m$, their product is an $m \times m$ diagonal matrix with entries $(\sigma_i)^2 = \lambda_i$, which are the eigenvalues of MM^*. □

4.3 Low Rank Approximation

We can safely ignore singular values that are equal to zero and, in doing so, we reduce the dimension of our matrix by ignoring the corresponding columns; effectively, we only care about the rank of the matrix which has the secondary effect of simplifying calculations. This is a first look at **dimension reduction**, which refers to reducing the dimension of matrices (or data in general) by ignoring things that don't matter.

The interpretation of small singular values as being insignificant is a shrewd one. Indeed, we may greatly reduce the complexity of calculations (thus speeding them up) if we simply ignore their corresponding singular vectors. This creates a matrix that is approximate to the one we began with,

[7] Or to at least use the widely available packages when the results matter; we always encourage pushing personal boundaries and limitations.

so how does this affect the accuracy of our calculation compared to the use the full version? This question has a concrete answer, given by the Eckart-Young-Mirsky Theorem[8]:

Theorem 4.3 (Eckart-Young-Mirsky)**.** *Let M be a matrix and $\|\cdot\|$ be a unitarily invariant norm. Then, for all matrices B such that* $\text{Rank}(B) = t$*, there exists a matrix M_t of rank t such that*

$$\|M - M_t\| \leq \|M - B\|$$

and we say that M_t is the ***low-rank approximation*** *of M. Moreover, if σ_i are singular values of M and* $\text{Rank}(M) = r$*,*

$$\|M - M_t\| = \begin{cases} \sigma_{t+1} & \text{for } \|\cdot\|_2 \\ \left(\sum_{i=t+1}^{r} \sigma_i^2\right)^{1/2} & \text{for } \|\cdot\|_F \end{cases}$$

In addition, the approximation of M by M_t is preserved by their respective SVDs:

$$M = U\Sigma V^* \approx U_t \Sigma_t V_t^* = M_t$$

Though the low-rank approximation exists, finding it is an entirely different matter and an active field of research - particularly for other matrix norms. We do have one more upshot! The proof that this holds for the Frobenius norm uses the SVD of M to show that $\|M - M_t\|_F^2$ is minimized; we can apply this to find *the* low-rank approximation under the Frobenius norm by simply truncating the smallest singular values, called **truncated SVD**. If we truncate after the t-th singular value, the optimal approximation is denoted

$$M \approx \widetilde{M} = U_t \Sigma_t V_t^*$$

[8]This is a generalization of the *matrix approximation lemma*, originally solved in 1906! At the time, this was proven only in the spectral norm; Mirsky generalized the result to arbitrary unitarily invariant norms, which allows us to use the Frobenius norm.

This is a link to the **Table of Contents**

Part III
Compression

This part explores the usage of the low-rank approximation in compression; specifically, we apply truncated SVD to accomplish this. However, it is worth noting that information is lost when this is done and that the process of undoing this is not fully-explored, as the algorithms required are not trivial.

5 Image Compression

In digital communication, optimization is of the utmost importance. The loading of pictures or videos in the early days of the internet was time consuming; today, since pictures and videos have higher quality, compression of files and images is essential to optimize data transmission efficiency. Because an image is fundamentally composed of pixels of three primary colors, any color image can be represented as three matrices corresponding to the color channels R, G, and B. In the following, we have a full-color image of the Grand Canyon.

Figure 1: Original

To illustrate the principles of image compression using the low-rank approximation, we will focus on grayscale images where the image is represented by a single matrix M containing the grayscale intensity levels of the pixels. As we apply truncated SVD, recall the definition of best rank-t approximation under the Frobenius norm and choose $t \in \{160, 40, 10\}$. The following pages are the results of the compression, and the program used is below.

This is a link to the **Table of Contents**

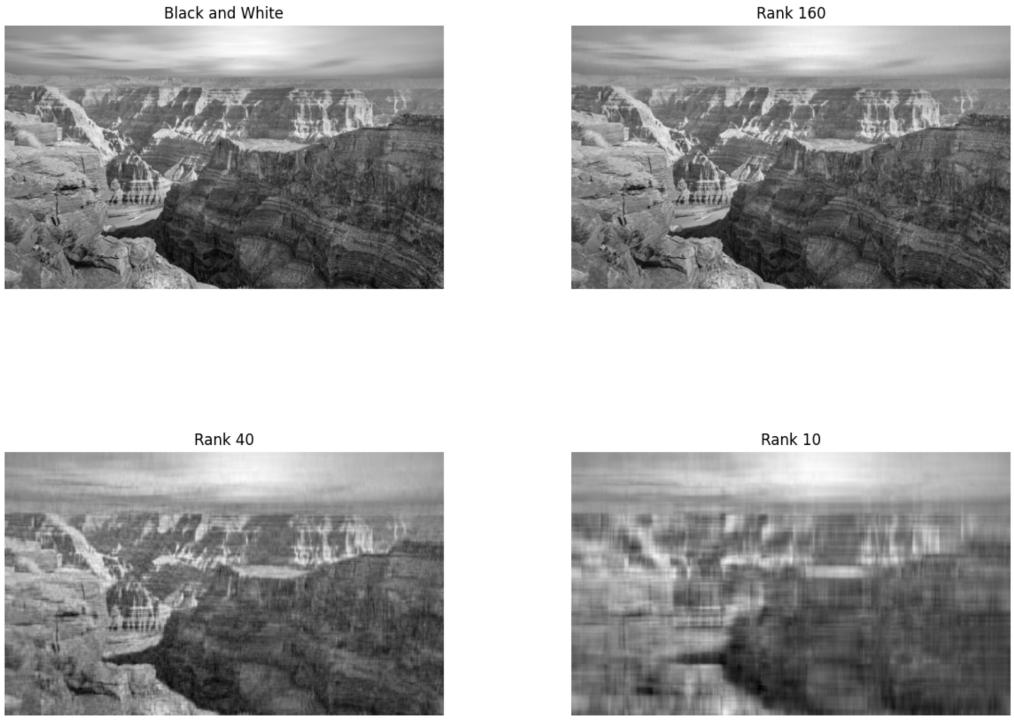


Figure 2: Various Compressions

6 Video Compression

Video compression is similar to image compression, extending to a much larger scale. Today, as videos reach higher resolutions, frame rates, and bit depths, efficient compression is essential for data transmission, storage, and real-time playback. Without it, modern applications such as video streaming, video conferencing, and mobile video sharing would be impractical due to excessive bandwidth and storage requirements.

A video is composed of a sequence of images played in rapid succession. Each frame is a matrix of pixel values, just like an image, but in video compression additional redundancies can be exploited. Spatial redundancy (within a frame) allows compression similar to images, while temporal redundancy (across frames) enables techniques that reduce redundant information between consecutive frames, such as SVD. We apply this method, which we used in image compression, to each frame of the video and note that the code took a long time to run. We strongly recommend *against* using this method to compress video data given the loss in quality is too great to be of any utility. While we lost fidelity with images, it is still possible to see what the image was supposed to resemble; however, the drop in quality combined with the loss of smoothness between frames makes the transformed video quite unpleasant to actually view.

This is a link to the **Table of Contents**

7 Sound Compression

Audio signals can be represented as matrices, typically in the form of spectrograms, where each column represents frequency content over time. SVD offers a powerful framework for analyzing and compressing such signals by isolating the most significant components of an audio waveform.

The spectrograms below illustrate how compression with truncated SVD affects the frequency content of an audio signal at different ranks. The original spectrogram (top left) shows the full resolution of the audio, with detailed harmonic structures and a wide frequency range. As the rank decreases, the reconstruction becomes increasingly blurred with high-frequency details gradually disappearing. At rank 5, much of the fine texture is lost, and the audio signal is reduced to its most dominant low-frequency components; thus, the decision to reduce the number of singular values in order to have better compression leads to a loss of quality and demonstrates the core limitations of simple low-rank approximations in signal processing.

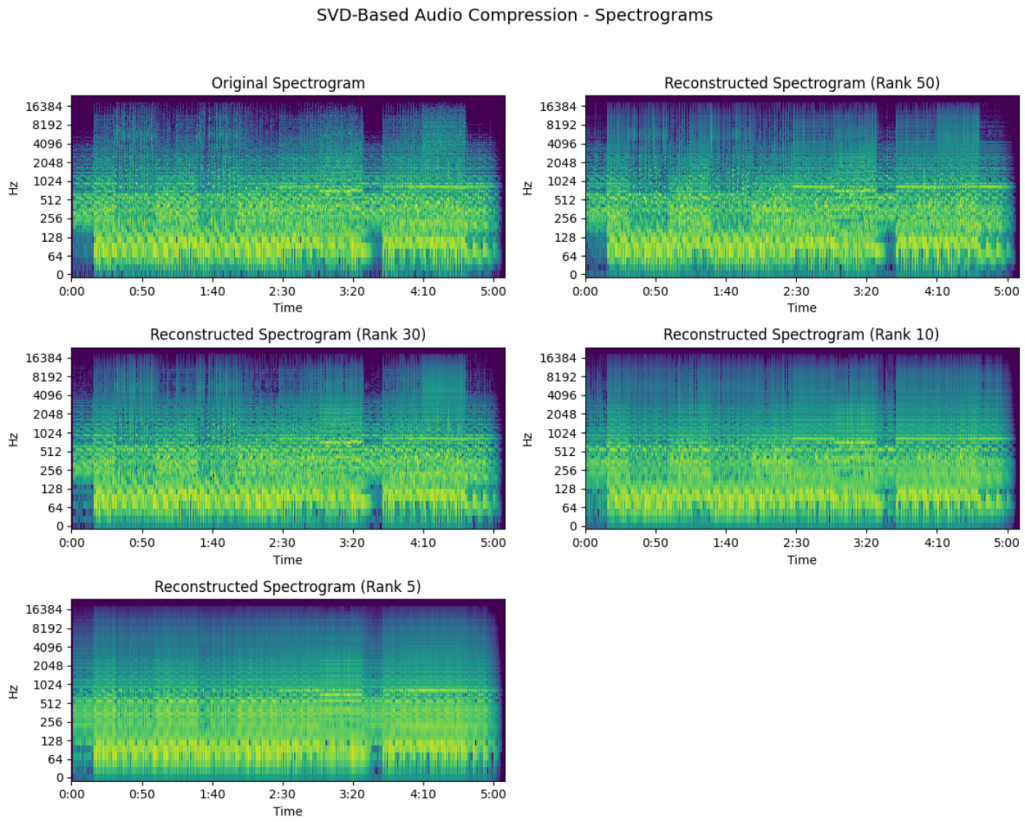


Figure 3: Spectrogram comparison

clocks_compressed_rank5	4/1/2025 10:17 PM	WAV File	28,873 KB
Coldplay_Clocks	4/1/2025 10:10 PM	WAV File	57,746 KB

Figure 4: Orginal file size vs compressed file size

8 Code

8.1 Image

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt

def compress_image(grandcannon, ranks=[160, 40, 10]):
    """
    Load an image, convert it to grayscale, perform SVD, and
    display the original plus compressed versions in separate windows.
    """
    # 1) Load and preprocess the image
    img = Image.open(grandcannon).convert('L')  # Convert to grayscale
    img = img.resize((img.width // 2, img.height // 2))  # Optional: downsample
    img_array = np.array(img) / 255.0  # Scale to [0,1]

    # 2) Compute the SVD
    U, S, Vt = np.linalg.svd(img_array, full_matrices=False)

    # 3) Display the original image in its own window
    plt.figure()
    plt.imshow(img_array, cmap='gray')
    plt.title('Black and White')
    plt.axis('off')

    # Show (non-blocking) so that subsequent windows can appear
    plt.show(block=False)

    # 4) For each rank k, reconstruct and display the compressed image
        in its own window
    for k in ranks:
        # Keep only the first k singular values
        Uk = U[:, :k]
        Sk = np.diag(S[:k])
        Vtk = Vt[:k, :]
        compressed_img = Uk @ Sk @ Vtk

        plt.figure()
        plt.imshow(compressed_img, cmap='gray')
        plt.title(f'Rank {k}')
        plt.axis('off')
        plt.show(block=False)

    # Finally, keep all figures open until you manually close them
    plt.show()

if __name__ == "__main__":
    compress_image("grandcannon.jpg", ranks=[160, 40, 10])
```

8.2 Video

```
import numpy as np
import imageio.v3 as imageio
import matplotlib.pyplot as plt


def compress_video(video_path="samplevideo.mp4", ranks=[160, 40, 10]):

# Read video frames
frames = imageio.imread(video_path, plugin="pyav")
# Reads all frames as a NumPy array
num_frames = frames.shape[0]

# Convert frames to grayscale and normalize
gray_frames = np.dot(frames[..., :3], [0.2989, 0.5870, 0.1140]) / 255.0

# Process only first 5 frames for preview
for frame_idx in range(min(5, num_frames)):
    frame = gray_frames[frame_idx]

# Perform SVD on the frame
U, S, Vt = np.linalg.svd(frame, full_matrices=False)

# Display original frame
plt.figure(figsize=(10, 5))
plt.subplot(1, len(ranks) + 1, 1)
plt.imshow(frame, cmap="gray")
plt.title(f"Frame {frame_idx} (Original)")
plt.axis("off")

# Reconstruct and display frames at different ranks
for i, k in enumerate(ranks):
    Uk, Sk, Vtk = U[:, :k], np.diag(S[:k]), Vt[:k, :]

# Low-rank approximation
compressed_frame = Uk @ Sk @ Vtk

plt.subplot(1, len(ranks) + 1, i + 2)
plt.imshow(compressed_frame, cmap="gray")
plt.title(f"Rank {k}")
plt.axis("off")
plt.show(block=False)

plt.show()

if __name__ == "__main__":
    compress_video(video_path="samplevideo.mp4", ranks=[160, 40, 10])
```

8.3 Sound

The program below was written using Python. The audio file is first converted into a spectrogram, then magnitude of the spectrogram is then decomposed into its singular values and vectors. By retaining only the top k singular values, the program reconstructs an approximation of the original audio signal by reducing its complexity while preserving its most prominent features. The compressed signal is then converted back to the time domain and saved as a new .wav file.

```
import numpy as np
import librosa
import librosa.display
import soundfile as sf
import matplotlib.pyplot as plt

def compress_audio_svd(audio_path, ranks=[50, 30, 10, 5],
        output_prefix="compressed_audio"):
    # Load audio file (mono)
    y, sr = librosa.load(audio_path, sr=None, mono=True)

    # Convert to spectrogram (STFT)
    S = librosa.stft(y)
    magnitude, phase = np.abs(S), np.angle(S)

    # Plot original spectrogram
    plt.figure(figsize=(12, 10))
    plt.subplot(3, 2, 1)
    librosa.display.specshow(librosa.amplitude_to_db(magnitude, ref=np.max),
                             sr=sr, y_axis='log', x_axis='time', cmap='viridis')
    plt.title("Original Spectrogram")

    for i, rank in enumerate(ranks):

        # Apply SVD to magnitude
        U, Sigma, Vt = np.linalg.svd(magnitude, full_matrices=False)

        # Truncate to rank-k
        U_k = U[:, :rank]
        Sigma_k = np.diag(Sigma[:rank])
        Vt_k = Vt[:rank, :]

        # Reconstruct magnitude
        magnitude_approx = U_k @ Sigma_k @ Vt_k
        S_approx = magnitude_approx * np.exp(1j * phase)

        # Inverse STFT
        y_reconstructed = librosa.istft(S_approx)

        # Save compressed audio
        output_path = f"{output_prefix}_rank{rank}.wav"
        sf.write(output_path, y_reconstructed, sr)
        print(f"Saved: {output_path}")
```

```
        # Plot spectrogram of the reconstructed signal
        plt.subplot(3, 2, i + 2)
        librosa.display.specshow(
            librosa.amplitude_to_db(
                magnitude_approx, ref=np.max),
                    sr=sr, y_axis='log', x_axis='time', cmap='viridis')
        plt.title(f"Reconstructed Spectrogram (Rank {rank})")

    plt.suptitle("SVD-Based Audio Compression - Spectrograms", fontsize=14)
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()

if __name__ == "__main__":
    compress_audio_svd("Coldplay_Clocks.wav", ranks=[50, 30, 10, 5],
    output_prefix="clocks_compressed")
```

Part IV
Principal Component Analysis

In the real world, taking measurements is not straightforward. Think of an engineer working to make durable materials - measuring a plate's ability to withstand an impact may be a worthy consideration, but its ability to retain its shape upon exposure to extreme temperature variations might be far more important. Perhaps neither of those are as important as its ability to maintain its strength upon repeated deformation. How do we determine what to measure to begin with? We might record data that we don't need and the noise associated with increasing the dimension of the data provides an excellent source of contamination that makes understanding relationships difficult. In fact, there is a name for this: the aptly-named "curse of dimensionality" refers to the computational and interpretive difficulties experienced as the dimension of data grows - namely, that the amount of data needed to accurately process grows as the dimensions (or features) increase, which necessitates a huge increase in computing power.

These problems can be resolved by either removing features directly or by extracting those that matter the most, the latter of which is the aim of principal component analysis (PCA). PCA has the benefit of being *nonparametric*, which is to say that any data set may have PCA applied to it without need for pre-processing. This is a massive upside in certain contexts but makes it wholly unusable in others, which are outlined when we discuss the assumptions we make when using PCA.

9 Statistics

Before diving into PCA, we need to introduce some statistics in the language of linear algebra. When a data set is represented as a matrix, our convention is that observations (or samples) are the columns with the variables' measurements given as rows.

Definition IV.1 (Sample Mean)**.** *Let X be a dataset with m random variables and n observations. The* ***sample mean*** *of the ith variable within the set is given by*

$$\bar{x}_i = \sum_{j=1}^{n} \frac{x_{ij}}{n}$$

So the sample mean $E[X]$ of X is the vector whose entries are the means of the observations for each variable:

$$E[X] = \begin{pmatrix} \bar{x}_1 \\ \vdots \\ \bar{x}_i \\ \vdots \\ \bar{x}_m \end{pmatrix}$$

For example, a sample X of men might include information about their income, age, height, and weight. After calculating, we would write this as the column vector

$$E[X] = \begin{pmatrix} 40,000 \\ 26.4 \\ 70.3 \\ 187.8 \end{pmatrix}$$

to represent the means of each category. This naturally leads us into the definition of the variance of X.

Definition IV.2 (Variance)**.** *For a set X of n observations, the **variance** of X is given by*

$$\text{var}(X) = \sigma_X^2 = \sum_{i=1}^{n} \frac{(x_i - E[X])^2}{n}$$

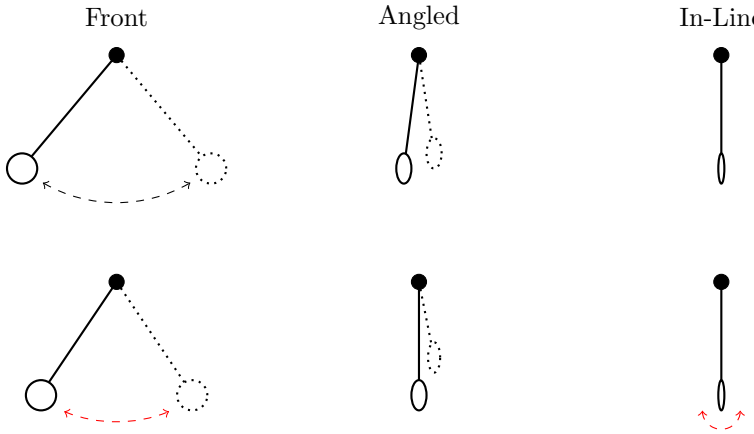
9.1 Signals, Noise, and Redundancy

When we take measurements, we are looking for signals that indicate a certain type of behavior and we measure this through variance. We must consider that we cannot take measurements in a vacuum because the real world exists and acts independently of our observations, meaning that we must account for its influence. In particular, a lack of precision in our tools will exacerbate this issue; that said, we can determine the usefulness of a measurement by the **signal to noise** ratio, which relates the variance due to the signal and the variance due to noise:

$$\text{SNR} = \frac{\sigma^2_{signal}}{\sigma^2_{noise}}$$

and $\text{SNR} \gg 1$ indicates that we have a high precision in our measurement. An otherwise low SNR indicates that our data is noisy, and likely not very useful.

Suppose we wanted to write an equation for the movement of a pendulum (in three dimensions) swinging back and forth by measuring the motion as captured by three cameras. Because this is the real world, the pendulum will not move perfectly in line with where we'd expect it to - air flow, friction, and a loose fit to its pivot can cause it to shake. On the following page is a visualization of the cameras' perspectives: the top row is what we would see with absolutely perfect knowledge, and the bottom row is what we may actually observe at any given moment, with the dash-dotted red lines indicate the variation in movement in a particular direction. The pendulum may not travel as expected, an angled view may differ between repititions, and there may be a wobble when we look in the direction of its motion. With sufficiently many observations, however, the variance due to noise will average to zero and we will be left with only the variance due to the signal - that is, we could actually write an equation for this pendulum.



9.2 Covariance

Covariance is a measure of the joint variability between two random variables (that is, the degree to which they are linearly related), so it follows that $\text{cov}(X, Y) = \text{cov}(Y, X)$; additionally, $\text{cov}(X, Y) > 0$ means that X and Y vary proportionally and $\text{cov}(X, Y) < 0$ means that X and Y vary inversely proportionally. Here is a formal definition:

Definition IV.3 (Covariance)**.** *If X and Y are random variables with n observations, their* ***covariance*** *is the expected value of the product of their variances:*

$$\begin{aligned}\operatorname{cov}(X,Y) = \sigma^2_{XY} &= E[(X - E[X])(Y - E[Y])] \\ &= \sum_{i=1}^{n} \frac{x_i y_i}{n}\end{aligned}$$

For sake of convenience, from here on we will assume that any given data set X has already been (column) centered by the calculation $X - E[X]$. If we interpret X and Y as row vectors, we may also interpret covariance as the mean of the following dot product:

$$\sigma^2_{XY} = E[XY^T]$$

Proposition 9.1. *If X and Y are independent random variables, then their covariance is zero.*

Proof. This fact follows from $E[XY] = E[X]E[Y]$[9]. □

Treating random variables as vectors allows us to apply familiar tools; in particular, we have the **covariance matrix**, denoted C_X, and with these tools come consequences that make our work significantly easier and are the backbone for PCA.

Definition IV.4 (Covariance Matrix)**.** *Let X be a data set as in Definition IV.1 with $k, \ell \leq m$. Then the matrix C_X is called the* ***covariance matrix****, where the entry $c_{k\ell}$ is given by $c_{k\ell} = \operatorname{cov}(x_k, x_\ell)$. In particular,*

$$C_X = \frac{1}{n} X X^T$$

Observation: Every covariance matrix C_X is symmetric and positive semi-definite.

Proof. It is obvious that C_X is symmetric as we multiply X and its transpose. Recall that a matrix M is positive semi-definite if and only if for any $x \in \mathbb{R}^n$, $x^T M x$ is not negative. Let x a component vector of X and let u be a real-valued column vector whose dimension is compatible with multiplication with X.

$$\begin{aligned}u^T C_X u &= u^T E[(x - \bar{x})(x - \bar{x})^T] u \\ &= E[u^T (x - \bar{x})(x - \bar{x})^T u] \\ &= E[s^2]\end{aligned}$$

where s is a zero-mean random scalar; thus, $E[s^2]$ is the square of the variance of s. Since $s \in \mathbb{R}$, it follows that $E[s^2] \geq 0$. □

Note 3. For square matrices over $\mathbb{R}$, we have the neat identity $X^T X = X X^T$. This is not true in general over $\mathbb{C}$ and it is never true when the dimensions disagree (e.g. 4×4 is different from a 5×5 matrix). This is why understanding the arrangement of the data is of the utmost importance! In general, it is expected that observations (samples) be columns with variables as rows.

Caution: When $E[XY] \approx E[X]E[Y]$, the equation $\operatorname{cov}(X,Y) = E[XY] - E[X]E[Y]$ is prone to catastrophic cancellation if $E[XY]$ and $E[X]E[Y]$ are not computed exactly. Centering the data avoids this!

[9] See section 18 here for details.

10 Principle Components

The basic idea of PCA is to take p variables, which may be correlated, and replace them with $k \leq p$ uncorrelated linear combinations of these variables; choosing the criteria for an optimal k is perhaps not straightforward, and PCA itself is prone to limitations. These limitations are made apparent by the necessary assumptions that we need to make, which we explore in the next section.

There is a change in terminology that we must also address. In statistics, what we call independent variables are called **features** in the context of PCA as well as machine learning (as these are the things that a machine learning algorithm will identify to actually "learn"). If we select k by explained variance, then what we are doing may be considered *feature extraction*.

Note 4. Determining whether X should be organized with observations as columns and variables/features as rows (our preference) or the other way around is a personal choice. The methods do not meaningfully change up to transposition.

10.1 Assumptions

As promised, here are the **core assumptions** that we are making when we do PCA:

1. The data is best described in linear relationships.
2. Measurements with the largest variance are the most important.
3. There is limited higher-order interaction between variables

Why these assumptions in particular? For the first assumption, perhaps most obvious, by assuming linearity we are able to use linear algebra and in particular the change of basis. Naturally, not all things in the real world are described or can be approximated (except perhaps locally) with linear mathematics. For example, the pendulum's motion is easiest to describe in polar coordinates; using PCA on data regarding its motion would not yield the best results. This issue can be resolved via a *kernel trick*, where we have a weighting function for a weighted sum or integral to embed nonlinear relationships into lower dimensions; this is incredibly useful for machine learning in particular. This is a *parametric* approach, utilizing PCA.

For the second point, when we assume that large variance is important, we also assume that the data has a high SNR. This may not be true as we might have variables unaccounted for! The third point is more difficult to resolve. PCA is able to deal with second-order dependencies, but this does not necessarily reveal all structure within the data - which may not be the case when we have a non-Gaussian distribution. In any of the cases outlined above it is necessary to carefully consider how the data should be treated and what other options for analysis there are. [3]

10.2 Overview

Now that we have explicitly stated our assumptions and considerations, let us outline our goals:

- Eliminate redundancy, which is to decorrelate the data set and is measured by covariance;
- Maximize signal, which is measured by variance;
- Identify the features and components that matter the most;
- Find the optimal transformation for the data set

Our covariance matrix C_X is almost assuredly to be nothing like this, but we may be able to manipulate it in order to make it happen. To be suggestive, let C_Y be the idealized form of C_X. Because it must be a diagonal matrix, we may apply the Spectral Theorem to find

$$C_X = PC_YP^T$$

hence

$$C_Y = P^T C_X P$$

We also want to arrange the entries of C_Y in a particular (descending) order along the diagonal in order to satisfy point 3 above, which amounts to reordering the eigenvectors of P^T as they correspond to the entries of C_Y. Doing this yields the **feature vector** F, whose columns are **principle components**. Thus we may finally say

$$C_Y = F C_X F^T$$

In other words, the eigenvectors of C_X *are* the principle components when we sort them by their eigenvalues in descending order. We should also point out that a limiting factor with PCA is that the interpretation of what any principle component represents is not straightforward. However, the contribution of a feature to a principle component is a measure of how strongly correlated it is to that feature, with higher absolute magnitudes indicating stronger contribution, so two features with high values in a principle component can be interpreted to vary together.

There are a few more terms that we need to define that are best done through example. Suppose that we measure the variables x, y, and z where we conduct PCA on the data set, and for our first principle component we have the linear combination

$$0.3x + 1.8y - 1.64z$$

Here, the **loadings** for P_1 are 0.3, 1.8, and -1.64 and the **PC score** corresponding to P_1 for a *particular* observation is the sum of the linear combination when we substitute x, y, and z for their corresponding observed values. Each loading in a principle component is an indication of how much each of the *original* features contributes to that particular principle component, which is the basis of conducting feature extraction - features that contribute little to any of the principle components can be safely ignored.

10.2.1 Simplified Calculation

Constructing C_X requires computing the product of a matrix and its transpose, which we know from Section 4.2.1 to be a risky endeavor. We claim that the the SVD of

$$Y = \frac{1}{\sqrt{n}} X^T$$

yields the principle components in the columns of V. Consider $Y^T Y$:

$$\begin{aligned} Y^T Y &= \left(\frac{1}{\sqrt{n}} X^T\right)^T \frac{1}{\sqrt{n}} X^T \\ &= \frac{1}{n} X X^T \\ &= C_X \end{aligned}$$

Thus by Proposition 4.1 we see that the columns of V are the eigenvectors of C_X (i.e., the principle components) and that we need not use eigenvalue decomposition to conduct PCA.

10.2.2 Feature Selection

PCA by itself does not select features, but we can measure how much a particular principle component matters and eliminate those that do not. The principle components are the axes along which

a data set has the most variance; because the eigenvalues sum to the total variance, we can express the ith principle component's contribution as the **explained variance ratio**, which is given by

$$EVR = \frac{\lambda_i}{\lambda_1 + \cdots + \lambda_n}$$

where λ_i is the eigenvalue for the ith principle component. Plotting the eigenvalues relative to their components generates a **scree plot**. The determination of how much total variance should be accounted for is subjective, as ignoring components means that we lose explanatory power as we reduce the dimension of the data.

The principle components are derived from the features of the original data set, so by examining the feature vector, we can choose to disregard the features that contribute little to the variance, which also reduces the dimension of the data. Doing this requires re-computing, but since this reduces the noise in the data, it may be worthwhile.

10.2.3 Implementation

In order to make this easier, here is a brief guide on how to go about implementing PCA. However, some details have been omitted to encourage careful reading and thought; in particular, constructing the feature vector and feature selection.

1. Normalize the data set to ensure a mean of zero and standard deviation of 1
2. Form the feature vector
   (a) Determine which principle components explain too little variance to be useful
   (b) Of the retained components, ignore the features that contribute little to them
3. Transform the data by the feature vector
4. Generate a biplot

10.3 Examples

For both of the following examples, we used python. In the first example, we coded the calculations for the covariance matrix by hand initially; in the second, we used packages specifically designed for PCA.

10.3.1 PCA of a Small Dataset

Given below, the matrix A represents the change in basis points of nine different bonds, issued by the US Treasury, over a period of five days where one basis point refers to a change in interest rate of 0.01%. For example, if a rate was at 4.5% today as opposed to 4% yesterday, we would say that the interest rate rose 50 basis points; if it was at 5% yesterday, it fell 50 basis points. It is our task to conduct PCA on A.

$$A = \begin{pmatrix} -5.4 & -19.4 & -12.4 & 19.6 & 17.6 \\ -4.6 & -14.6 & -12.6 & 14.4 & 17.4 \\ 1.0 & -14.0 & -14.0 & 9.0 & 18.0 \\ 9.6 & -10.4 & -16.4 & 2.6 & 14.0 \\ 13.4 & -10.6 & -17.6 & 1.4 & 13.4 \\ 18.6 & -11.4 & -15.4 & -0.4 & 8.6 \\ 20.8 & -11.2 & -14.2 & 0.8 & 3.8 \\ 20.8 & -12.2 & -11.2 & -0.2 & 2.8 \\ 14.6 & -7.4 & -7.4 & 0.6 & -0.4 \end{pmatrix}$$

First, note that A is a 9×5 matrix. This means that we have 9 features as the rows and 5 observations as the columns, meaning that we have quite a small sample size. We should transform A to be row-centered before we find the covariance matrix C_A.

$$C_A = \begin{pmatrix} 313.3 & 263.2 & 233.25 & 162.66 & 153.7 & 113.8 & 90.9 & 77.15 & 40.3 \\ 263.2 & 225.8 & 204.75 & 148.84 & 142.05 & 104.7 & 81.1 & 67.35 & 33.95 \\ 233.25 & 204.75 & 199.5 & 165.05 & 165.5 & 136.25 & 113 & 99.25 & 55 \\ 162.66 & 148.84 & 165.05 & 167.99 & 179.69 & 167.26 & 151.08 & 137.23 & 83.61 \\ 153.7 & 142.05 & 165.5 & 179.69 & 195.8 & 188.95 & 174.85 & 160.6 & 99.95 \\ 113.8 & 104.7 & 136.25 & 167.26 & 188.95 & 196.8 & 191.4 & 180.65 & 116.55 \\ 90.9 & 81.1 & 113 & 151.08 & 174.85 & 191.4 & 193.7 & 184.7 & 122.65 \\ 77.15 & 67.35 & 99.25 & 137.23 & 160.6 & 180.65 & 184.7 & 178.7 & 118.9 \\ 40.3 & 33.95 & 55 & 83.61 & 99.95 & 116.55 & 122.65 & 118.9 & 80.8 \end{pmatrix}$$

Now we decompose C_A using SVD. The singular values are as follows:

$$\sigma_1 = 1320.31, \quad \sigma_2 = 397.31, \quad \sigma_3 = 33.31, \quad \sigma_4 = 1.46,$$

We omit the other five singular values because they were too small; the difference between σ_4 and σ_5 is, logarithmically, comparable to the difference between the height of the average man and the width of a proton. In fact, we really only need to consider up to σ_2 given the disparity between it and σ_1. At any rate, the principal components are found as the columns of U. The first principal component is related to the weighted average of the daily change for all maturity lengths, the second adjusts the daily change's sensitivity to the maturity length of the bond, and the third is the change in the yield curve, which plots the interest rates (the yield) of bonds with differing maturities and whose slope predicts the direction of these interest rates. The interpretation here is that, perhaps unsurprisingly, shorter term bonds are more sensitive to changes in rates and that further changes make less of a difference to future changes for longer length bonds.

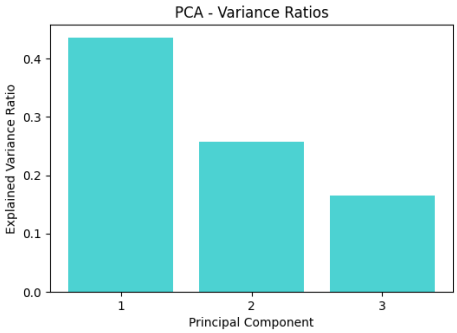
$$V^T = \begin{pmatrix} -0.38 & -0.34 & -0.36 & -0.35 & -0.37 & -0.35 & -0.32 & -0.3 & -0.18 \\ -0.53 & -0.44 & -0.26 & 0.03 & 0.13 & 0.29 & 0.36 & 0.38 & 0.28 \\ 0.48 & 0.04 & -0.23 & -0.44 & -0.44 & -0.12 & 0.23 & 0.35 & 0.36 \\ -0.07 & -0.2 & 0.51 & -0.16 & -0.23 & 0.2 & -0.45 & 0.57 & -0.23 \\ 0.12 & -0.22 & 0.2 & -0.25 & -0.29 & 0.64 & 0.29 & -0.48 & -0.16 \\ -0.04 & -0.23 & 0.49 & -0.51 & 0.43 & -0.41 & 0.29 & -0.09 & 0.06 \\ 0.16 & -0.5 & 0.3 & 0.57 & -0.36 & -0.33 & 0.14 & -0.1 & 0.2 \\ -0.53 & 0.55 & 0.3 & 0.02 & -0.45 & -0.14 & 0.34 & 0.02 & 0.02 \\ 0.14 & -0.08 & -0.19 & 0.09 & -0.01 & -0.17 & 0.44 & 0.26 & -0.8 \end{pmatrix}$$

$$U = \begin{pmatrix} -0.38 & -0.53 & 0.48 & -0.07 & -0.12 & 0.53 & -0.07 & 0.11 & -0.18 \\ -0.34 & -0.44 & 0.04 & -0.2 & 0.06 & -0.6 & -0.26 & -0.39 & 0.27 \\ -0.36 & -0.26 & -0.23 & 0.51 & -0.01 & -0.29 & 0.51 & 0.37 & -0.09 \\ -0.35 & 0.03 & -0.44 & -0.16 & 0.41 & 0.46 & 0.26 & -0.27 & 0.36 \\ -0.37 & 0.13 & -0.44 & -0.23 & 0.1 & -0.03 & -0.52 & 0.36 & -0.43 \\ -0.35 & 0.29 & -0.12 & 0.2 & -0.76 & 0.12 & -0.16 & -0.12 & 0.31 \\ -0.32 & 0.36 & 0.23 & -0.45 & -0.13 & -0.19 & 0.5 & -0.22 & -0.4 \\ -0.3 & 0.38 & 0.35 & 0.57 & 0.41 & -0.01 & -0.24 & -0.27 & -0.13 \\ -0.18 & 0.28 & 0.36 & -0.23 & 0.18 & -0.12 & 0 & 0.6 & 0.54 \end{pmatrix}$$

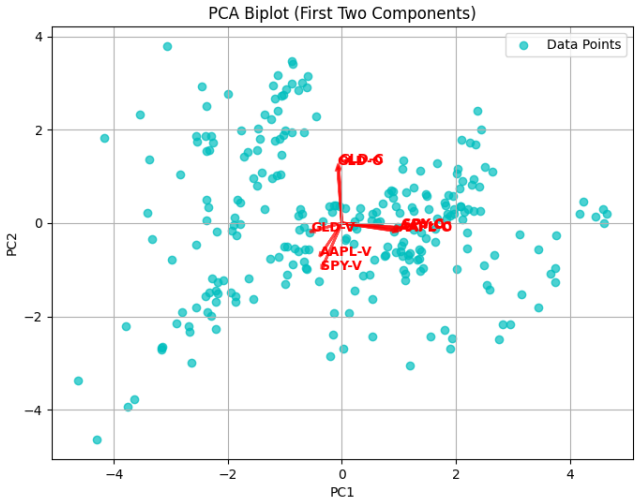
10.3.2 PCA of a Large Dataset

We used stock market data (Google Drive link to Excel spreadsheet) from December 30, 2020 to December 31, 2021 for AAPL, GLD, and SPY; specifically, the open and close prices as well as the trading volume. AAPL is the stock for Apple, GLD is an ETF tracking the price of gold, and SPY is an ETF tracking the performance of the S&P 500. Additionally, the code can be found here. Directly below, the PCs are given as rows while the columns are the weights of the open, close, and volume for AAPL, GLD, and SPY in that order. Immediately following is a chart of explained variance ratios; in order, they are are 0.4365, 0.2581, and 0.1657 which account for 86% of the variance in the data set.

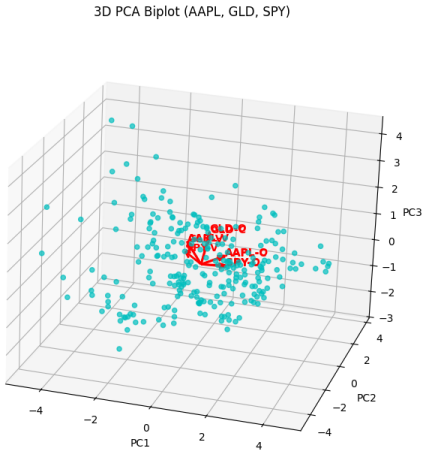
$$\begin{pmatrix} 0.46 & 0.46 & -0.18 & -0.04 & -0.03 & -0.25 & 0.48 & 0.48 & -0.17 \\ -0.08 & -0.08 & -0.32 & 0.57 & 0.58 & -0.08 & -0.05 & -0.04 & -0.46 \\ 0.26 & 0.24 & 0.53 & 0.39 & 0.35 & 0.4 & 0.06 & 0.04 & 0.39 \end{pmatrix}$$



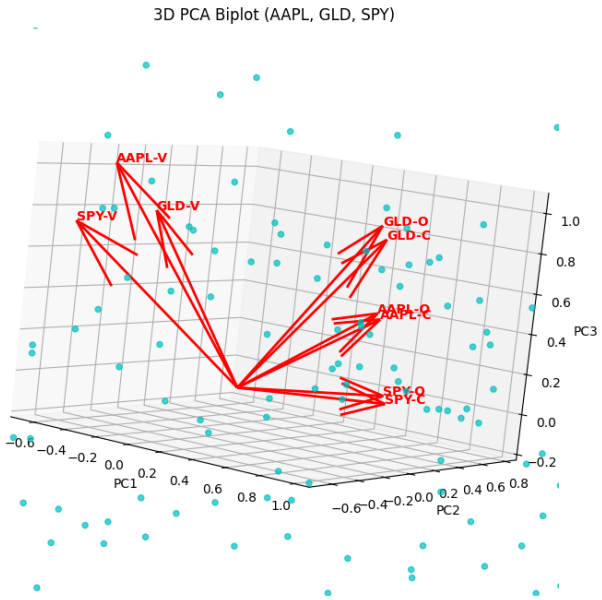
The following three figures are the data points that have been transformed by the PCA with the feature vectors overlaid, which serve to indicate correlation and relationships between stock prices and have been scaled by a factor of 2 to make them more visible. The first features only the first two components, but the *lack* of dimensionality makes it difficult to determine much. That said, it is quite apparent that the price of GLD is strongly correlated with PC2 but not PC1 while that of SPY and AAPL are moreso with PC1 but not PC2.



This is a link to the **Table of Contents**



These two images, above and below, are of the same scatterplot with the one below more zoomed in. Now that we have a third PC, it is apparent that the trading volume for each is more strongly correlated with PC3, somewhat with PC 2, while GLD has almost no correlation with PC1. PC2 is interesting: The price of GLD correlates strongly positively while the *volumes* of AAPL and SPY correlate negatively, with the strength of SPY being almost the same magnitude as GLD! The volume of GLD and the prices of AAPL and SPY, on the other hand, now appear to have no correlation with PC2 whatsoever.



11 Exercises

Exercise 2. Show that $\text{cov}(X, Y) = E[XY] - E[X]E[Y]$. **Hint**: As an operator, E is composed of addition and multiplication, which are both linear. Consider how E would interact with a mean.

Exercise 3. Find a counterexample to the converse of proposition 9.1. **Hint**: Define X such that $E[X] = 0$, then define Y relative to X. Think of intervals!

Exercise 4. Recall the matrix A of basis points of treasury bonds. The purpose of this exercise is to convince the reader that home-brewed algorithms are prone to error.

(a) Find C_A using a package as above, printing the result.

(b) Find C'_A using the definitions (i.e. don't use a package), printing the intermediate steps and result.

(c) Conducting PCA on both, print the following:

  (i) The matrices U, U', V^T, and V'^T

  (ii) σ_i and σ'_i for $i \leq 5$

(d) Let $C_A = M$ and $C'_A = B$. Verify the Perturbation Theorem for Singular Values and print the lower bound for the norm of E.

Part V
Code

12 Gram-Schmidt

```
import numpy as np

def gs_cofficient(v1, v2):
    return np.dot(v2, v1) / np.dot(v1, v1)

def multiply(cofficient, v):
    return map((lambda x : x * cofficient), v)

def proj(v1, v2):
    return multiply(gs_cofficient(v1, v2) , v1)

def gs(X):
    Y = []
    for i in range(len(X)):
        temp_vec = X[i]
        for inY in Y :
            proj_vec = proj(inY, X[i])
            #print "i =", i, ", projection vector =", proj_vec
            temp_vec = map(lambda x, y : x - y, temp_vec, proj_vec)
            #print "i =", i, ", temporary vector =", temp_vec
        Y.append(temp_vec)
    return Y


B = np.array([[5, 2, 1.5], [-1.0, -4.0, 1.0], [3.0, 0, 2.0]])


print np.array(gs(B))
```

13 PCA of a Small Dataset

Here is the code used in the PCA example. For our convenience, a section of code exports the resultant matrices into LaTeX.

```
import numpy as np
import pandas as pd
import xlrd
import matplotlib.pyplot as plt

# Import the correct file. Use r as a prefix or else it breaks
df = pd.read_excel(r"C:\Users\ebens\OneDrive\Documents\Treasury.xlsx")

matrix = df.to_numpy()

C = np.cov(matrix)


def matrix_to_latex(matrix: np.ndarray) -> str:
    """
    Converts a NumPy matrix into its LaTeX representation.

    Args:
        matrix (np.ndarray): The matrix to convert.

    Returns:
        str: A LaTeX string representation of the matrix.
    """
    rows, cols = matrix.shape
    latex_str = "\\begin{pmatrix}\n"

    for i in range(rows):
        row_values = [
            f"{matrix[i, j]:.2f}".rstrip('0').rstrip('.')
            if '.' in f"{matrix[i, j]:.2f}" else f"{matrix[i, j]:.2f}"
            for j in range(cols)
        ]
        latex_str += " & ".join(row_values) + " \\\\\n"

    latex_str += "\\end{pmatrix}"
    return latex_str



# Compute Singular Value Decomposition (SVD)
U, singular_values, V_T = np.linalg.svd(C)

# Output results
print("Covariance Matrix C:")
print(matrix_to_latex(C))

print("\nSingular Value Decomposition:")
```

```
print("U Matrix:")
print(matrix_to_latex(U))

print("\nV Transpose (V^T):")
print(matrix_to_latex(V_T))

print("\nSingular Values:")
print(singular_values)

# Plot Singular Values
plt.figure(figsize=(8, 5))
plt.plot(singular_values, marker='o', linestyle='-', label="Singular Values")
plt.xlabel("Index")
plt.ylabel("Singular Value")
plt.title("Singular Values of the Scaled Covariance Matrix")
plt.legend()
plt.grid(True)
plt.show()

# Heatmap of the Scaled Covariance Matrix
plt.figure(figsize=(8, 6))
plt.imshow(scaled_matrix, cmap='viridis', aspect='auto')
plt.colorbar(label="Value")
plt.title("Heatmap of the Scaled Covariance Matrix")
plt.xlabel("Column Index")
plt.ylabel("Row Index")
plt.show()
```

13.1 First Iteration

This is the first iteration of our code, as the title suggests. It is much more verbose and has a slightly different output to the above, which is attributable to rounding errors; the differences in results are not significant.

```
import numpy as np
import matplotlib.pyplot as plt
# Define the input matrix
x = np.array([
    [-5.4, -19.4, -12.4, 19.6, 17.6],
    [-4.6, -14.6, -12.6, 14.4, 17.4],
    [1, -14, -14, 9, 18],
    [9.6, -10.4, -16.4, 2.6, 14],
    [13.4, -10.6, -17.6, 1.4, 13.4],
    [18.6, -11.4, -15.4, -0.4, 8.6],
    [20.8, -11.2, -14.2, 0.8, 3.8],
    [20.8, -12.2, -11.2, -0.2, 2.8],
    [14.6, -7.4, -7.4, 0.6, -0.4]
])

def scale_matrix(matrix: np.ndarray, scalar: float) -> np.ndarray:
    """
    Scales a given matrix by a scalar.

    Args:
        matrix (np.ndarray): The input matrix.
        scalar (float): The scaling factor.

    Returns:
        np.ndarray: Scaled matrix.
    """
    return matrix * scalar

def matrix_to_latex(matrix: np.ndarray) -> str:
    """
    Converts a NumPy matrix into its LaTeX representation.

    Args:
        matrix (np.ndarray): The matrix to convert.

    Returns:
        str: A LaTeX string representation of the matrix.
    """
    rows, cols = matrix.shape
    latex_str = "\\begin{pmatrix}\n"

    for i in range(rows):
        row_values = [
            f"{matrix[i, j]:.2f}".rstrip('0').rstrip('.')
            if '.' in f"{matrix[i, j]:.2f}" else f"{matrix[i, j]:.2f}"
            for j in range(cols)
```

```
        ]
        latex_str += " & ".join(row_values) + " \\\\\n"

    latex_str += "\\end{pmatrix}"
    return latex_str

# Compute covariance-like matrix A * A^T
covariance_matrix = np.dot(x, x.T)

# Scale the matrix by a factor of 0.25
scalar = 0.25
scaled_matrix = scale_matrix(covariance_matrix, scalar)

# Compute Singular Value Decomposition (SVD)
U, singular_values, V_T = np.linalg.svd(scaled_matrix)

# Output results
print("\n--- LaTeX Representations ---\n")

print("Matrix A:")
print(matrix_to_latex(x))

print("\nTranspose of A:")
print(matrix_to_latex(x.T))

print("\nCovariance Matrix (A * A^T scaled by 0.25):")
print(matrix_to_latex(scaled_matrix))

print("\nSingular Value Decomposition:")
print("U Matrix:")
print(matrix_to_latex(U))

print("\nV Transpose (V^T):")
print(matrix_to_latex(V_T))

print("\nSingular Values:")
print(singular_values)
```

14 PCA of a Large Dataset

Here is the code for the large example.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from mpl_toolkits.mplot3d import Axes3D

file_path = r"C:\Users\ebens\OneDrive\Documents\stockdata.xlsx"


def matrix_to_latex(matrix: np.ndarray) -> str:
    """
    Converts a NumPy matrix into its LaTeX representation.

    Args:
        matrix(np.ndarray): The matrix to convert.
    Returns:
        str: A LaTeX string representation of the matrix.
    """
    rows, cols = matrix.shape
    latex_str = "\\begin{pmatrix}\n"

    for i in range(rows):
        row_values = [
            f"{matrix[i, j]:.2f}".rstrip('0').rstrip('.')
            if '.' in f"{matrix[i, j]:.2f}" else f"{matrix[i, j]:.2f}"
            for j in range(cols)
        ]
        latex_str += " & ".join(row_values) + " \\\\\n"

    latex_str += "\\end{pmatrix}"
    return latex_str

# Read the data for AAPL, GLD, and SPY
df_aapl = pd.read_excel(file_path, sheet_name=0, usecols=[1, 2, 3])
df_gld = pd.read_excel(file_path, sheet_name=1, usecols=[1, 2, 3])
df_spy = pd.read_excel(file_path, sheet_name=2, usecols=[1, 2, 3])

# Combine the datasets and standardize or else volume overwhelms calculation
df_combined = pd.concat([df_aapl, df_gld, df_spy], axis=1)
scaler = StandardScaler()
df_combined_scaled = scaler.fit_transform(df_combined)
pca = PCA(n_components=3)
df_combined_pca = pca.fit_transform(df_combined_scaled)

# Explained variance ratio
explained_variance_ratio = pca.explained_variance_ratio_
cumulative_variance = np.cumsum(explained_variance_ratio)
```

```
# Get the principal component vectors
pc_vectors = pca.components_
np.set_printoptions(precision=4, suppress=True)

# Print the matrix
print("Principal Component Vectors:\n", matrix_to_latex(np.matrix(pc_vectors)) )
print("Explained Variance Ratio", explained_variance_ratio)
print("Cumulative", cumulative_variance)

# Plot the explained variance
plt.figure(figsize=(6, 4))
plt.bar(range(1, 4), explained_variance_ratio, alpha=0.7, color='c')
plt.xticks(range(1, 4))
plt.ylabel("Explained Variance Ratio")
plt.xlabel("Principal Component")
plt.title("PCA - Variance Ratios")
plt.show()


# 2D Scatter Plot
plt.figure(figsize=(8, 6))
plt.scatter(df_combined_pca[:, 0], df_combined_pca[:, 1], alpha=0.7, color='c',
            label="Data Points")

# Get PCA loadings (feature vectors). 3 PCs, 9 original features
pc_vectors = pca.components_

# Define feature labels
feature_labels = ["AAPL-O", "AAPL-C", "AAPL-V", "GLD-O", "GLD-C", "GLD-V",
                    "SPY-O", "SPY-C", "SPY-V"]

# Scale vectors for better visualization
scaling_factor = 2
for i in range(len(feature_labels)):
    plt.arrow(0, 0,
              pc_vectors[0, i] * scaling_factor,
              pc_vectors[1, i] * scaling_factor,
              color='r', alpha=0.7, head_width=0.05, linewidth=2)
    plt.text(pc_vectors[0, i] * scaling_factor * 1.1,
             pc_vectors[1, i] * scaling_factor * 1.1,
             feature_labels[i], color='red', fontsize=10, fontweight='bold')

plt.xlabel("PC1")
plt.ylabel("PC2")
plt.title("PCA Biplot (First Two Components)")
plt.grid()
plt.legend()
plt.show()

# 3D Scatter plot
```

```
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(df_combined_pca[:, 0], df_combined_pca[:, 1],
           df_combined_pca[:, 2], alpha=0.7, color='c')

# Get PCA loadings (feature vectors). Shape (3,9) → 3 PCs, 9 original features
pc_vectors = pca.components_

# Define feature labels
feature_labels = ["AAPL-O", "AAPL-C", "AAPL-V", "GLD-O", "GLD-C", "GLD-V",
                  "SPY-O", "SPY-C", "SPY-V"]

# Scale the vectors
scaling_factor = 2
for i in range(len(feature_labels)):
    ax.quiver(0, 0, 0,
              pc_vectors[0, i] * scaling_factor,
              pc_vectors[1, i] * scaling_factor,
              pc_vectors[2, i] * scaling_factor,
              color='r', linewidth=2)
    ax.text(pc_vectors[0, i] * scaling_factor,
            pc_vectors[1, i] * scaling_factor,
            pc_vectors[2, i] * scaling_factor,
            feature_labels[i], color='red', fontsize=10, fontweight='bold')

ax.set_xlabel('PC1')
ax.set_ylabel('PC2')
ax.set_zlabel('PC3')
ax.set_title('3D PCA Biplot (AAPL, GLD, SPY)')
plt.show()
```

This is a link to the **Table of Contents**

Part VI
Bibliography

References

[1] Arnold, Mark E., *Sensitivity of Simple Eigenvalues*, https://arnold.hosted.uark.edu/NLA/Pages/EigCond.pdf. Accessed May 15, 2025.

[2] Intel, *Developer Reference for Intel oneAPI Math Kernel Library - Fortran*, https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-fortran/2024-0/overview.html. Accessed May 16, 2025.

[3] Shlens, *A Tutorial on Principal Component Analysis* (2014). ArXiv preprint, https://arxiv.org/pdf/1404.1100v1.